MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 602 November, 1980

Flavors: Message Passing in the Lisp Machine

Daniel Weinreb
David Moon

The object oriented programming style used in the Smalltalk and Actor languages is available in Lisp Machine Lisp, and used by the Lisp Machine software system. It is used to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter attempts to explain what programming with objects and with message passing means, the various means of implementing these in Lisp Machine Lisp, and when you should use them. It assumes no prior knowledge of any other languages.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-80-C-0505.

Keywords: Flavor, Message Passing, Actor, Smalltalk.

Table of Contents

Preface

This memo is intended to become a chapter in the Lisp Machine manual the next time it is published. Since there is a pressing need for documentation on flavors, we are publishing it immediately as a memo. The authors therefore assume that the reader has encountered this text while reading the manual. We assume that the reader is familiar with the basics of Lisp and the Lisp Machine's dialect in particular; we also make particular references to an example from section 17.1 in the manual.

Any comments, suggestions, or criticisms will be welcomed. The authors can be reached by any of the following communication paths:

ARPA Network mail to BUG-LMMAN@MIT-AI

U.S. Mail to
Daniel L. Weinreb or David A. Moon
545 Technology Square
Cambridge, Mass. 02139

Note

This document was edited with the Zmacs and Emacs editors, and formatted by the Bolio text justifier. It was printed on the MIT's Dover Printer.

1. Objects, Message Passing, and Flavors

1.1 Introduction

The object oriented programming style used in the Smalltalk and Actor families of languages is available in Lisp Machine Lisp, and used by the Lisp Machine software system. It is used to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter attempts to explain what programming with objects and with message passing means, the various means of implementing these in Lisp Machine Lisp, and when you should use them. It assumes no prior knowledge of any other languages.

1.2 Objects

When writing a program, it is often convenient to model what the program does in terms of *objects*: conceptual entities that can be likened to real-world things. Choosing what objects to provide in a program is very important to the proper organization of the program. In an object-oriented design, specifying what objects exist is the first task in designing the system. In a text editor, the objects might be "pieces of text", "pointers into text", and "display windows". In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows". After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object. In the text editor example, operations on "pieces of text" might include inserting text and deleting text; operations on "pointers into text" might include moving forward and backward; and operations on "display windows" might include redisplaying the window and changing with which "piece of text" the window is associated.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it. More rigorously, the program defines several *types* of object (the editor above has three types), and it can create many *instances* of each type (that is, there can be many pieces of text, many pointers into text, and many windows). The program defines a set of types of object, and the operations that can be performed on any of the instances of each type.

This should not be wholly unfamiliar to the reader. Earlier in this manual, we saw a few examples of this kind of programming. A simple example is disembodied property lists, and the functions get, putprop, and remprop. The disembodied property list is a type of object; you can instantiate one with (cons nil nil) (that is, by evaluating this form you can create a new disembodied property list); there are three operations on the object, namely get, putprop, and remprop. Another example in the manual was the first example of the use of defstruct, which was called a ship. defstruct automatically defined some operations on this object: the operations to access its elements. We could define other functions that did useful things with ships, such as computing their speed, angle of travel, momentum, or velocity, stopping them, moving them elsewhere, and so on.

In both cases, we represent our conceptual object by one Lisp object. The Lisp object we use for the representation has *structure*, and refers to other Lisp objects. In the property list case, the Lisp object is a list with alternating indicators and values; in the ship case, the Lisp object is an array whose details are taken care of by defstruct. In both cases, we can say that the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. get examines the state of a property list, and putprop alters it; ship-x-position and ship-get-momentum examine the state of a ship, and (setf (ship-mass) 5.0) and (ship-move-to 3.0 4.0) alter it.

We have now seen the essence of object-oriented programming. A conceptual object is modelled by a single Lisp object, which bundles up some state information. For every type of object, there is a set of operations that can be performed to examine or alter the state of the object.

1.3 Modularity

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. If you have modular programming constructs and techniques availabile, it helps and encourages you to write programs that are easy to read and understand, and so are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller with a set of external interfaces, without requiring the caller to understand how the internal details of the implementation work. In other words, a program that calls this facility can treat the facility as a black box; the program knows what the facility's external interfaces guarantee to do, and that is all it knows.

For example, a program that uses disembodied property lists never needs to know that the property list is being maintained as a list of alternating indicators and values; the program simply performs the operations, passing them inputs and getting back outputs. The program only depends on the external definition of these operations: it knows that if it putprops a property, and doesn't remprop it (or putprop over it), then it can do get and be sure of getting back the same thing it put in. The important thing about this hiding of the details of the implementation is that someone reading a program that uses disembodied property lists need not concern himself with how they are implemented; he need only understand what they undertake to do. This saves the programmer a lot of time, and lets him concentrate his energies on understanding the program he is working on. Another good thing about this hiding is that the representation of property lists could be changed, and the program would continue to work. For example, instead of a list of alternating elements, the property list could be implemented as an association list or a hash table. Nothing in the calling program would change at all.

The same is true of the ship example. The caller is presented with a collection of operations, such as ship-x-position, ship-y-position, ship-speed, and ship-direction; it simply calls these and looks at their answers, without caring how they did what they did. In our example above, ship-x-position and ship-y-position would be accessor functions, defined automatically by defstruct, while ship-speed and ship-direction would be functions defined by the implementor of the ship type. The code might look like this:

```
(defstruct (ship)
  ship-x-position
  ship-y-position
  ship-x-velocity
  ship-y-velocity
  ship-mass)

(defun ship-speed (ship)
  (sqrt (+ (^ (ship-x-velocity ship) 2)
           (^ (ship-y-velocity ship) 2))))

(defun ship-direction (ship)
  (atan (ship-y-velocity ship)
        (ship-x-velocity ship)))
```

The caller need not know that the first two functions were structure accessors and that the second two were written by hand and do arithmetic. Those facts would not be considered part of the black box characteristics of the implementation of the ship type. The ship type does not guarantee which functions will be implemented in which ways; such aspects are not part of the contract between ship and its callers. In fact, ship could have been written this way instead:

```
(defstruct (ship)
  ship-x-position
  ship-y-position
  ship-speed
  ship-direction
  ship-mass)

(defun ship-x-velocity (ship)
  (* (ship-speed ship) (cos (ship-direction ship))))

(defun ship-y-velocity (ship)
  (* (ship-speed ship) (sin (ship-direction ship))))
```

In this second implementation of the ship type, we have decided to store the velocity in polar coordinates instead of rectangular coordinates. This is purely an implementation decision; the caller has no idea which of the two ways the implementation works, because he just performs the operations on the object by calling the appropriate functions.

We have now created our own types of objects, whose implementations are hidden from the programs that use them. Such types are usually referred to as *abstract types*. The object-oriented style of programming can be used to create abstract types by hiding the implementation of the operations, and simply documenting what the operations are defined to do.

Some more terminology: the quantities being held by the elements of the ship structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides identity (eqness)) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; our two ways of writing the ship operations have different

instance variables, but from the outside they have exactly the same operations.

One might ask: "But what if the caller evaluates (aref ship 3) and notices that he gets back the x-velocity rather than the speed? Then he can tell which of the two implementations were used." This is true; if the caller were to do that, he could tell. However, when a facility is implemented in the object-oriented style, only certain functions are documented and advertised: the functions which are considered to be operations on the type of object. The contract from ship to its callers only speaks about what happens if the caller calls these functions. The contract makes no guarantees at all about what would happen if the caller were to start poking around on his own using aref. A caller who does so *is in error*; he is depending on something that is not specified in the contract. No guarantees were ever made about the results of such action, and so anything may happen; indeed, ship may get reimplemented overnight, and the code that does the aref will have a different effect entirely and probably stop working. This example shows why the concept of a contract between a callee and a caller is important: the contract is what specifies the interface between the two modules.

Unlike some other languages that provide abstract types, Lisp Machine Lisp makes no attempt to have the language automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that the Lisp Machine is an interactive system, and so it is important to be able to examine and alter internal state interactively (usually from a debugger). Furthermore, there is no strong distinction between the "system" programs and the "user" programs on the Lisp Machine; users are allowed to get into any part of the language system and change what they want to change.

In summary: by defining a set of operations, and making only a specific set of external entrypoints available to the caller, the programmer can create his own abstract types. These types can be useful facilities for other programs and programmers. Since the implementation of the type is hidden from the callers, modularity is maintained, and the implementation can be changed easily.

We have hidden the implementation of an abstract type by making its operations into functions which the user may call. The important thing is not that they are functions—in Lisp everything is done with functions. The important thing is that we have defined a new conceptual operation and given it a name, rather than requiring anyone who wants to do the operation to write it out step-by-step. Thus we say (ship-x-velocity s) rather than (aref s 2).

It is just as true of such abstract-operation functions as of ordinary functions that sometimes they are simple enough that we want the compiler to compile special code for them rather than really calling the function. (Compiling special code like this is often called *open-coding*.) The compiler is directed to do this through use of macros, defsubsts, or optimizers. defstruct arranges for this kind of special compilation for the functions that get the instance variables of a structure.

When we use this optimization, the implementation of the abstract type is only hidden in a certain sense. It does not appear in the Lisp code written by the user, but does appear in the compiled code. The reason is that there may be some compiled functions that use the macros (or whatever); even if you change the definition of the macro, the existing compiled code will continue to use the old definition. Thus, if the implementation of a module is changed programs that use it may need to be recompiled. This is something we sometimes accept for the sake of

efficiency.

In the present implementation of flavors, which is discussed below, there is no such compiler incorporation of nonmodular knowledge into a program, except when the "outside-accessible instance variables" feature is used; see page 22, where this problem is explained further. If you don't use the "outside-accessible instance variables" feature, you don't have to worry about this.

1.4 Generic Operations

Suppose we think about the rest of the program that uses the ship abstraction. It may want to deal with other objects that are like ships in that they are movable objects with mass, but unlike ships in other ways. A more advanced model of a ship might include the concept of the ship's engine power, the number of passengers on board, and its name. An object representing a meteor probably would not have any of these, but might have another attribute such as how much iron is in it.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object the attributes apply to. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship will need to know the ship's attributes, and will have to call `ship-x-position` and `ship-y-velocity` and so on. The problem is that these functions won't work for meteors. There would have to be a second program to calculate orbits for meteors that would be exactly the same, except that where the first one calls `ship-x-position`, the second one would call `meteor-x-position`, and so on. This would be very bad; a great deal of code would have to exist in multiple copies, all of it would have to be maintained in parallel, and it would take up space for no good reason.

What is needed is an operation that can be performed on objects of several different types. For each type, it should do the thing appropriate for that type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in most programming languages, including Lisp Machine Lisp. The `+` (or `plus`) function will accept either fixnums or flonums, and perform either fixnum addition or flonum addition, whichever is appropriate, based on the data types of the objects being manipulated. In our example, we need a generic `x-position` operation that can be performed on either ships, meteors, or any other kind of mobile object represented in the system. This way, we can write a single program to calculate orbits. When it wants to know the *x* position of the object it is dealing with, it simply invokes the generic `x-position` operation on the object, and whatever type of object it has, the correct operation is performed, and the *x* position is returned.

A terminology for the use of such generic operations has emerged from the Smalltalk and Actor languages: performing a generic operation is called *sending a message*. The objects in the program are thought of as little people, who get sent messages and respond with answers. In the example above, the objects are sent `x-position` messages, to which they respond with their *x* position. This *message passing* is how generic operations are performed.

Sending a message is a way of invoking a function. Along with the *name* of the message, in general, some arguments are passed; when the object is done with the message, some values are returned. The sender of the message is simply calling a function with some arguments, and getting some values back. The interesting thing is that the caller did not specify the name of a procedure to call. Instead, it specified a message name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke was found from this information.

When a message is sent to an object, a function therefore must be found to handle the message. The two data used to figure out which function to call are the *type* of the object, and the *name* of the message. The same set of functions are used for all instances of a given type, so the type is the only attribute of the object used to figure out which function to call. The rest of the message besides the name are data which are passed as arguments to the function, so the name is the only part of the message used to find the function. Such a function is called a *method*. For example, if we send an x-position message to an object of type ship, then the function we find is "the ship type's x-position method". A method is a function that handles a specific kind of message to a specific kind of object; this method handles messages named x-position to objects of type ship.

In our new terminology: the orbit-calculating program finds the *x* position of the object it is working on by sending that object a message named x-position (with no arguments). The returned value of the message is the *x* position of the object. If the object was of type ship, then the ship type's x-position method was invoked; if it was of type meteor, then the meteor type's x-position method was invoked. The orbit-calculating program just sends the message, and the right function is invoked based on the type of the object. We now have true generic functions, in the form of message passing; the same operation can mean different things depending on the type of the object.

1.5 Generic Operations in Lisp

How do we implement message passing in Lisp? By convention, objects that receive messages are always *functional* objects (that is, you can apply them to arguments), and a message is sent to an object by calling that object as a function, passing the name of the message as the first argument, and the arguments of the message as the rest of the arguments. Message names are represented by symbols; normally these symbols are in the keyword package (see chapter 19 of the Lisp Machine Manual) since messages are a protocol for communication between different programs, which may reside in different packages. So if we have a variable my-ship whose value is an object of type ship, and we want to know its *x* position, we send it a message as follows:

```
(funcall my-ship ':x-position)
```

This form returns the *x* position as its returned value. To set the ship's *x* position to 3.0, we send it a message like this:

```
(funcall my-ship ':set-x-position 3.0)
```

It should be stressed that no new features are added to Lisp for message sending; we simply define a convention on the way objects take arguments. The convention says that an object accepts messages by always interpreting its first argument as a message name. The object must

consider this message name, find the function which is the method for that message name, and invoke that function.

This raises the question of how message receiving works. The object must somehow find the right method for the message it is sent. Furthermore, the object now has to be callable as a function; objects can't just be **defstructs** any more, since those aren't functions. But the structure defined by **defstruct** was doing something useful: it was holding the instance variables (the internal state) of the object. We need a function with internal state; that is, we need a coroutine.

Of the Lisp Machine Lisp features presented so far, the most appropriate is the closure (see chapter 10 of the Lisp Machine Manual). A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a big **selectq** form to dispatch on its first argument. (Actually, rather than using closures and a **selectq**, the Lisp Machine provides *entities* and **defselect**; see page 30.)

While using closures (or entities) does work, it has several serious problems. The main problem is that in order to add a new operation to a system, it is necessary to modify a lot of code; you have to find all the types that understand that operation, and add a new clause to the **selectq**. The problem with this is that you cannot textually separate the implementation of your new operation from the rest of the system; the methods must be interleaved with the other operations for the type. Adding a new operation should only require *adding* Lisp code; it should not require *modifying* Lisp code.

The conventional way of making generic operations is to have a procedure for each operation, which has a big **selectq** for all the types; this means you have to modify code to add a type. The way described above is to have a procedure for each type, which has a big **selectq** for all the operations; this means you have to modify code to add an operation. Neither of these has the desired property that extending the system should only require adding code, rather than modifying code.

Closures (and entities) are also somewhat clumsy and crude. A far more streamlined, convenient, and powerful system for creating message-receiving objects exists; it is called the *Flavor* mechanism. With flavors, you can add a new method simply by adding code, without modifying anything. Furthermore, many common and useful things to do are very easy to do with flavors. The rest of this chapter describes flavors.

1.6 Simple Use of Flavors

A *flavor*, in its simplest form, is a definition of an abstract type. New flavors are created with the **defflavor** special form, and methods of the flavor are created with the **defmethod** special form. New instances of a flavor are created with the **make-instance** function. This section explains simple uses of these forms.

For an example of a simple use of flavors, here is how the **ship** example above would be implemented.

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables)

(defmethod (ship :speed) ()
  (sqrt (+ (^ x-velocity 2)
           (^ y-velocity 2))))

(defmethod (ship :direction) ()
  (atan y-velocity x-velocity))
```

The code above creates a new flavor. The first subform of the defflavor is ship, which is the name of the new flavor. Next is the list of instance variables; they are the five that should be familiar by now. The next subform is something we will get to later. The rest of the subforms are the body of the defflavor, and each one specifies an option about this flavor. In our example, there is only one option, namely :gettable-instance-variables. This means that for each instance variable, a method should automatically be generated to return the value of that instance variable. The name of the message is a symbol with the same name as the instance variable, but interned on the keyword package. Thus, methods are created to handle the messages :x-position, :y-position and so on.

Each of the two defmethod forms adds a method to the flavor. The first one adds a handler to the flavor ship for messages named :speed. The second subform is the lambda-list, and the rest is the body of the function that handles the :speed message. The body can refer to or set any instance variables of the flavor, the same as it can with local variables or special variables. When any instance of the ship flavor is invoked with a first argument of :direction, the body of the second defmethod will be evaluated in an environment in which the instance variables of ship refer to the instance variables of this instance (the one to which the message was sent). So when the arguments of atan are evaluated, the values of instance variables of the object to which the message was sent will be used as the arguments. atan will be invoked, and the result it returns will be returned by the instance itself.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor will have the five instance variables named in the defflavor form, and the seven methods we have seen (five that were automatically generated because of the :gettable-instance-variables option, and two that we wrote ourselves). The way to create an instance of our new flavor is with the make-instance function. Here is how it could be used:

```
(setq my-ship (make-instance 'ship))
```

This will return an object whose printed representation is:

```
#<SHIP 13731210>
```

(Of course, the value of the magic number will vary; it is not interesting anyway.) The argument to make-instance is, as you can see, the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*, that is, commands to the flavor of which we are making an instance, selecting optional features. This will be discussed more in a moment.

Examination of the flavor we have defined shows that it is quite useless as it stands, since there is no way to set any of the parameters. We can fix this up easily, by putting the :settable-instance-variables option into the defflavor form. This option tells defflavor to generate methods for messages named :set-x-position, :set-y-position, and so on; each such method takes one argument, and sets the corresponding instance variable to the given value.

Another option we can add to the defflavor is :initable-instance-variables, to allow us to initialize the values of the instance variables when an instance is first created. :initable-instance-variables does not create any methods; instead, it makes *initialization keywords* named :x-position, :y-position, etc., that can be used as init-option arguments to make-instance to initialize the corresponding instance variables. The set of init options are sometimes called the *init-plist* because they are like a property list.

Here is the improved defflavor:

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

All we have to do is evaluate this new defflavor, and the existing flavor definition will be updated and now include the new methods and initialization options. In fact, the instance we generated a while ago will now be able to accept these new messages! We can set the mass of the ship we created by evaluating

```
(funcall my-ship ':set-mass 3.0)
```

and the mass instance variable of my-ship will properly get set to 3.0. If you want to play around with flavors, it is useful to know that describe of an instance tells you the flavor of the instance and the values of its instance variables. If we were to evaluate (describe my-ship) at this point, the following would be printed:

```
#<SHIP 13731210>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:             unbound
        Y-POSITION:             unbound
        X-VELOCITY:             unbound
        Y-VELOCITY:             unbound
        MASS:                   3.0
```

Now that the instance variables are "initable", we can create another ship and initialize some of the instance variables using the init-plist. Let's do that and describe the result:

```
(setq her-ship (make-instance 'ship ':x-position 0.0
                                    ':y-position 2.0
                                    ':mass 3.5))
                ==> #<SHIP 13756521>

(describe her-ship)
#<SHIP 13756521>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:             0.0
        Y-POSITION:             2.0
        X-VELOCITY:             unbound
        Y-VELOCITY:             unbound
        MASS:                   3.5
```

A flavor can also establish default initial values for instance variables. These default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list, whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example:

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(setq another-ship (make-instance 'ship ':x-position 3.4))

(describe another-ship)
#<SHIP 14563643>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:             3.4
        Y-POSITION:             0.0
        X-VELOCITY:             2.0
        Y-VELOCITY:             3.0
        MASS:                   unbound
```

x-position was initialized explicitly, so the default was ignored. y-position was initialized from the default value, which was 0.0. The two velocity instance variables were initialized from their default values, which came from two global variables. mass was not explicitly initialized and did not have a default initialization, so it was left unbound.

There are many other options that can be used in defflavor, and the init options can be used more flexibly than just to initialize instance variables; full details are given later in this chapter. But even with the small set of features we have seen so far, it is easy to write object-oriented programs.

1.7 Mixing Flavors

Now we have a system for defining message-receiving objects so that we can have generic operations. If we want to crea e a new type called meteor that would accept the same generic operations as ship, we could simply write another defflavor and two more defmethods that looked just like those of ship, and then meteors and ships would both accept the same operations. ship would have some more instance variables for holding attributes specific to ships, and some more methods for operations that are not generic, but are only defined for ships; the same would be true of meteor.

However, this would be a a wasteful thing to do. The same code has to be repeated in several places, and several instance variables have to be repeated. The code now needs to be maintained in many places, which is always undesirable. The power of flavors (and the name "flavors") comes from the ability to mix several flavors and get a new flavor. Since the functionality of ship and meteor partially overlap, we can take the common functionality and move it into its own flavor, which might be called moving-object. We would define moving-object the same way as we defined ship in the previous section. Then, ship and meteor could be defined like this:

```
(defflavor ship (engine-power number-of-passengers name)
                (moving-object)
   :gettable-instance-variables)

(defflavor meteor (percent-iron) (moving-object)
   :initable-instance-variables)
```

These defflavor forms use the second subform, which we ignored previously. The second subform is a list of flavors to be combined to form the new flavor; such flavors are called *components*. Concentrating on ship for a moment (analogous things are true of meteor), we see that it has exactly one component flavor: moving-object. It also has a list of instance variables, which includes only the ship-specific instance variables and not the ones that it shares with meteor. By incorporating moving-object, the ship flavor acquires all of its instance variables, and so need not name them again. It also acquires all of moving-object's methods, too. So with the new definition, ship instances will still accept the :x-velocity and :speed messages, and they will do the same thing. However, the :engine-power message will also be understood (and will return the value of the engine-power instance variable).

What we have done here is to take an abstract type, moving-object, and build two more specialized and powerful abstract types on top of it. Any ship or meteor can do anything a moving object can do, and each also has its own specific abilities. This kind of building can continue; we could define a flavor called ship-with-passenger that was built on top of ship, and it would inherit all of moving-object's instance variables and methods as well as ship's instance variables and methods. Furthermore, the second subform of defflavor can be a list of

several components, meaning that the new flavor should combine all the instance variables and methods of all the flavors in the list, as well as the ones *those* flavors are built on, and so on. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. We sometimes use the term "components" to mean the immediate components (the ones listed in the defflavor), and sometimes to mean all the components (including the components of the immediate components and so on). (Actually, it is not strictly a tree, since some flavors might be components through more than one path. It is really a directed graph; it can even be cyclic.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including non-terminal nodes *before* the subtrees they head, and eliminating duplicates. For example, if flavor-1's immediate components are flavor-2 and flavor-3, and flavor-2's components are flavor-4 and flavor-5, and flavor-3's component was flavor-4, then the complete list of components of flavor-1 would be:

```
flavor-1, flavor-2, flavor-4, flavor-5, flavor-3
```

The flavors earlier in this list are the more specific, less basic ones; in our example, ship-with-passengers would be first in the list, followed by ship, followed by moving-object. A flavor is always the first in the list of its own components. Notice that flavor-4 does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done during the walk; if there is a cycle in the directed graph, it will not cause a non-terminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both flavor-2 and flavor-3 have instance variables named foo, then flavor-1 will have an instance variable named foo, and any methods that refer to foo will refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. (Typically, only one component ever sets the variable, and the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each message supported by the flavor. This function is constructed out of all the methods for that message from all the components of the flavor. There are many different ways that methods can be combined; these can be selected by the user when a flavor is defined. The user can also create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods we have seen are *primary* methods. The default way primary methods are combined is that all but the earliest one provided are ignored. In other words, the combined method is simply the primary method of the first flavor to provide a primary method. What this means is that if you are starting with a flavor foo and building a flavor bar on top of it, then you can override foo's method for a message by providing your own method. Your method will be called, and foo's will never be called.

Simple overriding is often useful; if you want to make a new flavor bar that is just like foo except that it reacts completely differently to a few messages, then this will work. However, often you don't want to completely override the base flavor's (foo's) method; sometimes you want to add some extra things to be done. This is where combination of methods is used.

The usual way methods are combined is that one flavor provides a primary method, and other flavors provide *daemon methods*. The idea is that the primary method is "in charge" of the main business of handling the message, but other flavors just want to keep informed that the message was sent, or just want to do the part of the operation associated with their own area of responsibility.

When methods are combined, a single primary method is found; it comes from the first component flavor that has one. Any primary methods belonging to later component flavors are ignored. This is just what we saw above; bar could override foo's primary method by providing its own primary method.

However, you can defined other kinds of methods. In particular, you can defined *daemon* methods. They come in two kinds, *before* and *after*. There is a special syntax in defmethod for defining such methods. Here is an example of the syntax. To give the ship flavor an after-daemon method for the :speed message, the following syntax would be used:

```
(defmethod (ship :after :speed) ()
  body)
```

Now, when a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before daemons, then the primary method, then all the after daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. Before-daemons are called in the order that flavors are combined, while after-daemons are called in the reverse order. In other words, if you build bar on top of foo, then bar's before-daemons will run before any of those in foo, and bar's after-daemons will run after any of those in foo.

The reason for this order is to keep the modularity order correct. If we create flavor-1 built on flavor-2; then it should not matter what flavor-2 is built out of. Our new before-daemons go before all those of flavor-2, and our new after-daemons go after all those of flavor-2. Note that if you have no daemons, this reduces to the form of combination described above. The most recently added component flavor is the highest level of abstraction; you build a higher-level object on top of a lower-level object by adding new components to the front. The syntax for defining daemon methods can be found in the description of defmethod below.

To make this a bit more clear, let's consider a simple example that is easy to play with: the :print-self method. The Lisp printer (i.e. the print function; see sections 18.2 and 18.4 in the Lisp Machine Manual) prints instances of flavors by sending them :print-self messages. The first argument to the :print-self message is a stream (we can ignore the others for now), and the receiver of the message is supposed to print its printed representation on the stream. In the ship example above, the reason that instances of the ship flavor printed the way they did is because the ship flavor was actually built on top of a very basic flavor called vanilla-flavor; this component is provided automatically by defflavor. It was vanilla-flavor's :print-self method that was doing the printing. Now, if we give ship its own primary method for the :print-self message, then that method will take over the job of printing completely; vanilla-flavor's method will not be called at all. However, if we give ship a before-daemon method for the :print-self message, then it will get invoked before the vanilla-flavor message, and so whatever it prints will appear before what vanilla-flavor prints. So we can use before-daemons to add prefixes to a printed representation; similarly, after-daemons can add suffixes.

There are other ways to combine methods besides daemons, but this way is the most common. The more advanced ways of combining methods are explained in a later section; see page 25. The vanilla-flavor and what it does for you are also explained later; see page 24.

1.8 Flavor Functions

defflavor *Macro*

A flavor is defined by a form

(defflavor *flavor-name* (*var1 var2...*) (*flav1 flav2...*)
opt1 opt2...)

flavor-name is a symbol which serves to name this flavor. It will get an si:flavor property of the internal data-structure containing the details of the flavor.

(typep *obj*), where *obj* is an instance of the flavor named *flavor-name*, will return the symbol *flavor-name*. (typep *obj flavor-name*) is t if *obj* is an instance of a flavor, one of whose components (possibly itself) is *flavor-name*.

var1, *var2*, etc. are the names of the instance-variables containing the local state for this flavor. A list of the name of an instance-variable and a default initialization form is also acceptable; the initialization form will be evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable will remain unbound.

flav1, *flav2*, etc. are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.

opt1, *opt2*, etc. are options; each option may be either a keyword symbol or a list of a keyword symbol and arguments. The options to defflavor are described on page 20.

all-flavor-names *Variable*

This is a list of the names of all the flavors that have ever been defflavor'ed.

defmethod *Macro*

A method, that is, a function to handle a particular message sent to an instance of a particular flavor, is defined by a form such as

(defmethod (*flavor-name method-type message*) *lambda-list*
form1 form2...)

flavor-name is a symbol which is the name of the flavor which is to receive the method. *method-type* is a keyword symbol for the type of method; it is omitted when you are defining a primary method, which is the usual case. *message* is a keyword symbol which names the message to be handled.

The meaning of the *method-type* depends on what kind of method-combination is declared for this message. For instance, for daemons :before and :after are allowed. See page 25 for a complete description of method types and the way methods are combined.

lambda-list describes the arguments and "aux variables" of the function; the first argument to the method, which is the message keyword, is automatically handled, and so it is not included in the *lambda-list*. Note that methods may not have &quole arguments; that is

they must be functions, not special forms. *form1*, *form2*, etc. are the function body; the value of the last form is returned.

The variant form

(defmethod (*flavor-name message*) *function*)

where *function* is a symbol, says that *flavor-name*'s method for *message* is *function*, a symbol which names a function. That function must take appropriate arguments; the first argument is the message keyword.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, a message name, and a method type, there can only be one function, so if you define a :before daemon method for the foo flavor to handle the :bar message, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, message name or flavor.

defmethod actually defines a symbol, called the *flavor-method-symbol*, as a function, and the flavor system goes through that symbol to call the method. Sometimes it is useful to deal with such a symbol: for example, you can use it to trace a method with trace (see page 252 in the Lisp Machine Manual). The flavor-method-symbol is formed by concatenating (with hyphens) the flavor name, the method type, the message name, and "method" (for example, ship-x-position-method, ship-after-y-velocity-method, ship-combined-mass-method, etc.).

make-instance *flavor-name init-option1 value1 init-option2 value2...*

Creates and returns an instance of the specified flavor. Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options, as described above. If the flavor supports the :init message, it is sent to the newly-created object with one argument, the init-plist. This is a disembodied property-list containing the init-options specified and those defaulted from the flavor's :default-init-plist. make-instance is an easy-to-call interface to instantiate-flavor; for full details refer to that function.

instantiate-flavor *flavor-name init-plist* &optional *send-init-message-p return-unhandled-keywords area*

This is an extended version of make-instance, giving you more features. Note that it takes the init-plist as an argument, rather than taking a &rest argument of init-options and values.

The *init-plist* argument must be a disembodied property list; locf of a &rest argument will do. Beware! This property list can be modified; the properties from the default-init-plist are putprop'ed on if not already present.

In the event that :init methods do remprop of properties already on the init-plist (as opposed to simply doing get and putprop), then the init-plist will get rplacd'ed. This means that the actual list of options will be modified. It also means that locf of a &rest argument will not work; the caller of instantiate-flavor must copy its rest argument (e.g. with append); this is because rplacd is not allowed on &rest arguments.

First, if the flavor's method-table and other internal information have not been computed or are not up to date, they are computed. This may take a substantial amount of time and invoke the compiler, but will only happen once for a particular flavor no matter how many instances you make, unless you change something.

Next, the instance variables are initialized. There are several ways this initialization can happen. If an instance variable is declared initable, and a keyword with the same spelling as its name appears in *init-plist*, it is set to the value specified after that keyword. If an instance variable does not get initialized this way, and an initialization form was specified for it in a **defflavor**, that form is evaluated and the variable is set to the result. The initialization form may not depend on any instance variables nor on **self**; it will not be evaluated in the "inside" environment in which methods are called. If an instance variable does not get initialized either of these ways it will be left unbound; presumably an **:init** method should initialize it (see below). Note that a simple empty disembodied property list is **(nil)**, which is what you should give if you want an empty init-plist. If you use **nil**, the property list of **nil** will be used, which is probably not what you want.

If any keyword appears in the *init-plist* but is not used to initialize an instance variable and is not declared in an **:init-keywords** option (see page 20) it is presumed to be a misspelling. If the *return-unhandled-keywords* argument is not supplied, such keywords are complained about by signalling an error. But if *return-unhandled-keywords* is supplied non-**nil**, a list of such keywords is returned as the second value of **instantiate-flavor**.

Note that default values in the *init-plist* can come from the **:default-init-plist** option to **defflavor**. See page 20.

If the *send-init-message-p* argument is supplied and non-**nil**, an **:init** message is sent to the newly-created instance, with one argument, the *init-plist*. **get** can be used to extract options from this property-list. Each flavor that needs initialization can contribute an **:init** method, by defining a daemon.

If the *area* argument is specified, it is the number of an area in which to cons the instance; otherwise it is consed in the default area.

defwrapper *Macro*

This is hairy and if you don't understand it you should skip it.

Sometimes the way the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In that case **defwrapper** can be used to define a macro which expands into code which is wrapped around the invocation of the methods. This is best explained by an example; suppose you needed a lock locked during the processing of the **:foo** message to the **bar** flavor, which takes two arguments, and you have a **lock-frobboz** special-form which knows how to lock the lock (presumably it generates an **unwind-protect**). **lock-frobboz** needs to see the first argument to the message; perhaps that tells it what sort of operation is going to be performed (read or write).

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  `(lock-frobboz (self arg1)
     . ,body))
```

The use of the body macro-argument prevents the defwrapper'ed macro from knowing the exact implementation and allows several defwrappers from different flavors to be combined properly.

Note well that the argument variables, arg1 and arg2, are not referenced with commas before them. These may look like defmacro "argument" variables, but they are not. Those variables are not found at the time the defwrapper-defined macro is expanded and the back-quoting is done; rather the result of that macro-expansion and back-quoting is code which, when a message is sent, will bind those variables to the arguments in the message as local variables of the combined method.

Consider another example. Suppose you thought you wanted a :before daemon, but found that if the argument was nil you needed to return from processing the message immediately, without executing the primary method. You could write a wrapper such as

```
(defwrapper (bar :foo) ((arg1) . body)
  `(cond ((null arg1))              ;Do nothing if arg1 is nil
         (t before-code
            . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular message; perhaps the :after daemons need to know what the primary method did, and it is something that cannot be easily deduced from just the arguments. You might use an instance variable for this, or you might create a special variable which is bound during the processing of the message and used free by the methods.

```
(defvar *communication*)
(defwrapper (bar :foo) (ignore . body)
  `(let ((*communication* nil))
     . ,body))
```

Similarly you might want a wrapper which puts a *catch around the processing of a message so that any one of the methods could throw out in the event of an unexpected condition.

If you change a wrapper, the change may not take effect automatically. You must use recompile-flavor with a third argument of nil to force the effect to propagate into the compiled code which the system generates to implement the flavor. The reason for this is that the flavor system cannot reliably tell the difference between reloading a file containing a wrapper and really redefining the wrapper to be different, and propagating a change to a wrapper is expensive. [This may be fixed in the future.]

Like daemon methods, wrappers work in outside-in order; when you add a defwrapper to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. However, *all* wrappers happen before *any* daemons happen. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new

flavors will execute within that wrapper's context.

self *Variable*

When a message is sent to an object, the variable **self** is automatically bound to that object, for the benefit of methods which want to manipulate the object itself (as opposed to its instance variables).

funcall-self *message arguments...*

When **self** is an instance or an entity, (**funcall-self** *args...*) has the same effect as (**funcall self** *args...*) except that it is a little faster since it doesn't have to re-establish the context in which the instance variables evaluate correctly. If **self** is not an instance (nor an "entity", see page 30), **funcall-self** and **funcall self** do the same thing.

When **self** is an instance, **funcall-self** will only work correctly if it is used in a method or a function, wrapped in a **declare-flavor-instance-variables**, that was called from a method. Otherwise the instance-variables will not be already set up.

lexpr-funcall-self *message arguments... list-of-arguments*

This function is a cross between **lexpr-funcall** and **funcall-self**. When **self** is an instance or an entity, (**lexpr-funcall-self** *args...*) has the same effect as (**lexpr-funcall self** *args...*) except that it is a little faster since it doesn't have to re-establish the context in which the instance variables evaluate correctly. If **self** is not an instance (nor an "entity", see page 30), **lexpr-funcall-self** and **lexpr-funcall** do the same thing.

declare-flavor-instance-variables *Macro*

Sometimes you will write a function which is not itself a method, but which is to be called by methods and wants to be able to access the instance variables of the object **self**. The form

```
(declare-flavor-instance-variables (flavor-name)
  function-definition)
```

surrounds the *function-definition* with a declaration of the instance variables for the specified flavor, which will make them accessible by name. Currently this works by declaring them as special variables, but this implementation may be changed in the future. Note that it is only legal to call a function defined this way while executing inside a method for an object of the specified flavor, or of some flavor built upon it.

recompile-flavor *flavor-name* &optional *single-message* *(use-old-combined-methods* **t***)* *(do-dependents* **t***)*

Updates the internal data of the flavor and any flavors that depend on it. If *single-message* is supplied non-nil, only the methods for that message are changed. The system does this when you define a new method that did not previously exist. If *use-old-combined-methods* is **t**, then the existing combined method functions will be used if possible. New ones will only be generated if the set of methods to be called has changed. This is the default. If *use-old-combined-methods* is **nil**, automatically-generated functions to call multiple methods or to contain code generated by wrappers will be regenerated unconditionally. If you change a wrapper, you must do **recompile-flavor** with third argument **nil** in order to make the new wrapper take effect. If *do-dependents* is **nil**, only the specific flavor you specified will be recompiled. Normally it and all flavors that depend on it will be recompiled.

recompile-flavor only affects flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins (see page 23).

compile-flavor-methods *Macro*

The form (compile-flavor-methods *flavor-name-1 flavor-name-2...*), placed in a file to be compiled, will cause the compiler to include the automatically generated combined methods for the named flavors in the resulting qfasl file, provided all of the necessary flavor definitions have been made. Use of compile-flavor-methods for all flavors that are going to be instantiated is recommended to eliminate the need to call the compiler at run time (the compiler will still be called if incompatible changes have been made, such as addition or deletion of methods that must be called by a combined method).

get-handler-for *object message*

Given an object and a message, will return that object's method for that message, or nil if it has none. When *object* is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method. If you get back a combined method, you can use the List Combined Methods editor command (page 31) to find out what it does.

This function can be used with other things than flavors, and has an optional argument which is not relevant here.

symeval-in-instance *instance symbol* &optional *no-error-p*

This function is used to find the value of an instance variable inside a particular instance. *Instance* is the instance to be examined, and *symbol* is the instance variable whose value should be returned. If there is no such instance variable, an error is signalled, unless *no-error-p* is non-nil in which case nil is returned.

set-in-instance *instance symbol value*

This function is used to alter the value of an instance variable inside a particular instance. *Instance* is the instance to be altered, *symbol* is the instance variable whose value should be set, and *value* is the new value. If there is no such instance variable, an error is signalled.

si:describe-flavor *flavor-name*

This function prints out descriptive information about a flavor; it is self-explanatory. An important thing it tells you that can be hard to figure out yourself is the combined list of component flavors; this list is what is printed after the phrase "and directly or indirectly depends on".

si:*flavor-compilations*

This variable contains a history of when the flavor mechanism invoked the compiler. It is a list; elements toward the front of the list represent more recent compilations. Elements are typically of the form

(:method *flavor-name type message-name*)

and *type* is typically :combined.

You may **setq** this variable to **nil** at any time; for instance before loading some files that you suspect may have missing or obsolete **compile-flavor-methods** in them.

1.9 Defflavor Options

There are quite a few options to **defflavor**. They are all described here, although some are for very specialized purposes and not of interest to most users. Each option can be written in two forms; either the keyword by itself, or a list of the keyword and "arguments" to that keyword.

Several of these options declare things about instance variables. These options can be given with arguments which are instance variables, or without any arguments in which case they refer to all of the instance variables listed at the top of the **defflavor**. This is *not* necessarily all the instance variables of the component flavors; just the ones mentioned in this flavor's **defflavor**. When arguments are given, they must be instance variables that were listed at the top of the **defflavor**; otherwise they are assumed to be misspelled and an error is signalled. It is legal to declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

:gettable-instance-variables

Enables automatic generation of methods for getting the values of instance variables. The message name is the name of the variable, in the keyword package (i.e. put a colon in front of it.)

:settable-instance-variables

Enables automatic generation of methods for setting the values of instance variables. The message name is ":set-" followed by the name of the variable. All settable instance variables are also automatically made gettable and **initable**.

:initable-instance-variables

The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *initable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to **make-instance**.

:init-keywords

The arguments are declared to be keywords in the initialization property-list which are processed by this flavor's **:init** methods. This is just used by error-checking which looks for entries (presumably misspelled) in the initialization property-list which are not handled by any component flavor of the object being created, neither as initable-instance-variables nor as init-keywords.

:default-init-plist

The arguments are alternating keywords and value forms, like a property-list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example,

```
(:default-init-plist :frob-array
                     (make-array nil 'art-q 100))
```

would provide a default "frob array" for any instance for which the user did not provide one explicitly.

:required-instance-variables

Declares that any flavor incorporating this one which is instantiated into an object must contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those which checks the spelling of its arguments in the way described at the start of this section.

Required instance variables may be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the defflavor is that the latter declares that this flavor "owns" those variables and will take care of initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-methods

The arguments are names of messages which any flavor incorporating this one must handle. An error occurs if there is an attempt to instantiate such a flavor and it is lacking a method for one of these messages. Typically this option appears in the defflavor for a base flavor (see page 23).

:included-flavors

The arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the defflavor is that when component flavors are combined, all the included flavors come after all the regular flavors. Thus included flavors act like defaults. For an example of the use of included flavors, consider the ship example given earlier, and suppose we want to define a relativity-mixin which increases the mass dependent on the speed. We might write,

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (// mass (sqrt (- 1 (^ (// (funcall-self ':speed)
                             *speed-of-light*)
                         2)))))
```

but this would lose because any flavor that had relativity-mixin as a component would get moving-object right after it in its component list. As a base flavor, moving-object should be last in the list of components so that other components mixed in can replace its methods and so that daemon methods combine in the right order. So instead we write,

```
(defflavor relativity-mixin () ()
           (:included-flavors moving-object))
```

which allows relativity-mixin's methods to access moving-object instance variables such as mass (the rest mass), but does not specify a place for moving-object in the list of components. (Actually it puts it at the end, where it will usually be eliminated as a duplicate.)

:no-vanilla-flavor

Unless this option is specified, si:vanilla-flavor is included (in the sense of the :included-flavors option). vanilla-flavor provides some default methods for the :print-self, :describe, :which-operations, :get-handler-for, :eval-inside-yourself, and :funcall-inside-yourself messages. See page 24.

:default-handler

The argument is the name of a function which is to be called when a message is received

for which there is no method. It will be called with whatever arguments the instance was called with, including the message name; whatever values it returns will be returned. If this option is not specified on any component flavor, it defaults to a function which will signal an error.

:ordered-instance-variables

This option is mostly for esoteric internal system uses. The arguments are names of instance variables which must appear first (and in this order) in all instances of this flavor, or any flavor depending on this flavor. This is used for instance variables which are specially known about by microcode, and in connection with the :outside-accessible-instance-variables option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this defflavor.

:outside-accessible-instance-variables

The arguments are instance variables which are to be accessible from "outside" of this object, that is from functions other than methods. A macro (actually a defsubst) is defined which takes an object of this flavor as an argument and returns the value of the instance variable; setf may be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are similar to the accessor macros created by defstruct (see chapter 17 of the Lisp Machine Manual.)

This feature works in two different ways, depending on whether the instance variable has been declared to have a fixed slot in all instances, via the :ordered-instance-variables option.

If the variable is not ordered, the position of its value cell in the instance will have to be computed at run time. This takes noticeable time, although less than actually sending a message would take. An error will be signalled if the argument to the accessor macro is not an instance or is an instance which does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor the accessor macro was defined for, or a flavor built upon that flavor. This error check would be too expensive.

If the variable is ordered, the compiler will compile a call to the accessor macro into a subprimitive which simply accesses that variable's assigned slot by number. This subprimitive is only 3 or 4 times slower than car. The only error-checking performed is to make sure that the argument is really an instance and is really big enough to contain that slot. There is no check that the accessed slot really belongs to an instance variable of the appropriate name. Any functions that use these accessor macros will have to be recompiled if the number or order of instance variables in the flavor is changed. The system will not know automatically to do this recompilation. If you aren't very careful, you may forget to recompile something, and have a very hard-to-find bug. Because of this problem, and because using these macros is less elegant than sending messages, the use of this option is discouraged. In any case the use of these accessor macros should be confined to the module which owns the flavor, and the "general public" should send messages.

:select-method-order

This is purely an efficiency hack due to the fact that currently the method-table is

searched linearly when a message is sent. The arguments are names of messages which are frequently used or for which speed is important. Their methods are moved to the front of the method table so that they are accessed more quickly.

:method-combination

Declares the way that methods from different flavors will be combined. Each "argument" to this option is a list (*type order message1 message2...*). *Message1*, *message2*, etc. are names of messages whose methods are to be combined in the declared fashion. *type* is a keyword which is a defined type of combination; see page 25. *Order* is a keyword whose interpretation is up to *type*; typically it is either :base-flavor-first or :base-flavor-last.

Any component of a flavor may specify the type of method combination to be used for a particular message. If no component specifies a type of method combination, then the default type is used, namely :daemon. If more than one component of a flavor specifies it, then they must agree on the specification, or else an error is signalled.

:documentation

The list of arguments to this option is remembered on the flavor's property list as the :documentation property. The (loose) standard for what can be in this list is as follows; this may be extended in the future. A string is documentation on what the flavor is for; this may consist of a brief overview in the first line, then several paragraphs of detailed documentation. A symbol is one of the following keywords:

:mixin A flavor that you may want to mix with others to provide a useful feature.

:essential-mixin
A flavor that must be mixed in to all flavors of its class, or inappropriate behavior will ensue.

:lowlevel-mixin
A mixin used only to build other mixins.

:combination A combination of flavors for a specific purpose.

:special-purpose
A flavor used for some internal or kludgey purpose by a particular program, which is not intended for general use.

This documentation can be viewed with the si:describe-flavor function (see page 19) or the editor's Meta-X Describe Flavor command (see page 30).

1.10 Flavor Families

The following organization conventions are recommended for all programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which will have that base flavor as one of their components. Typically the base flavor includes things relevant to the whole family, such as instance variables, :required-methods and :required-instance-variables declarations, default methods for certain messages, :method-combination declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most are not instantiatable and merely serve as a base upon

which to build other flavors. The base flavor for the *foo* family is often named *basic*-foo.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them, along with the appropriate base flavor. By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies would be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named *mumble*-mixin.

If you are writing a program that uses someone else's facility to do something, using that facility's flavors and methods, your program might still define its own flavors, in a simple way. The facility might provide a base flavor and a set of mixins, and the caller can combine these in various combinations depending on exactly what it wants, since the facility probably would not provide all possible useful combinations. Even if your private flavor has exactly the same components as a pre-existing flavor, it can still be useful since you can use its :default-init-plist (see page 20) to select options of its component flavors and you can define one or two methods to customize it "just a little".

1.11 Vanilla flavor

Unless you specify otherwise (with the :no-vanilla-flavor option to defflavor), every flavor *includes* the "vanilla" flavor, which has no instance variables but provides some basic useful methods. Thus, nearly every instance may be assumed to handle the following messages.

:print-self *stream prindepth slashify-p*

The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance or an entity. The arguments are the stream, the current depth in list-structure (for comparison with prinlevel), and whether slashification is enabled (prin1 vs princ; see page 154 in the Lisp Machine Manual). Vanilla-flavor ignores the last two arguments, and prints something like #<*flavor-name octal-address*>. The *flavor-name* tells you what type of object it is, and the *octal-address* allows you to tell different objects apart (provided the garbage collector doesn't move them behind your back).

:describe

The object should describe itself, printing a description onto the standard-output stream. The describe function sends this message when it encounters an instance or an entity. Vanilla-flavor outputs the object, the name of its flavor, and the names and values of its instance-variables, in a reasonable format.

:which-operations
The object should return a list of the messages it can handle. Vanilla-flavor generates the list once per flavor and remembers it, minimizing consing and compute-time. If a new method is added, the list is regenerated the next time someone asks for it.

:get-handler-for *operation*
The object should return the method it uses to handle *operation*. If it has no handler for that message, it should return nil. This is like the get-handler-for function (see page 19), but, of course, you can only use it on objects known to accept messages.

:eval-inside-yourself *form*
The argument is a form which is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to setq one of these special variables; the instance variable will be modified. This is mainly intended to be used for debugging. An especially useful value of *form* is (break t); this gets you a Lisp top level loop inside the environment of the methods of the flavor, allowing you to examine and alter instance variables, and run functions that use the instance variables.

:funcall-inside-yourself *function* &rest *args*
function is applied to *args* in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to setq one of these special variables; the instance variable will be modified. This is mainly intended to be used for debugging.

1.12 Method Combination

As was mentioned earlier, there are many ways to combine methods. The way we have seen is called the :daemon type of combination. To use one of the others, you use the :method-combination option to defflavor (see page 23) to say that all the methods for a certain message to this flavor, or a flavor built on it, should be combined in a certain way.

The following types of method combination are supplied by the system. It is possible to define your own types of method combination; for information on this, see the code. Note that for most types of method combination other than :daemon you must define the order in which the methods are combined, either :base-flavor-first or :base-flavor-last. In this context, base-flavor means the last element of the flavor's fully-expanded list of components.

Which method type keywords are allowed depends on the type of method combination selected. Many of them allow only untyped methods. There are also certain method types used for internal purposes.

:daemon — This is the default type of method combination. All the :before methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the :after methods are called. The value returned is the value of the primary method.

:progn — All the methods are called, inside a progn special form. No typed methods are allowed. This means that all of the methods are called, and the result of the combined method is whatever the last of the methods returns.

:or — All the methods are called, inside an or special form. No typed methods are allowed. This means that each of the methods is called in turn. If a method returns a non-nil value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it doesn't want to handle the message, it should return nil, and the next method will get a chance to try.

:and — All the methods are called, inside an and special form. No typed methods are allowed. The basic idea is much like :or; see above.

:list — Calls all the methods and returns a list of their returned values. No typed methods are allowed.

:inverse-list — Calls each method with one argument; these arguments are successive elements of the list which is the sole argument to the message. No typed methods are allowed. Returns no particular value. If the result of a :list-combined message is sent back with an :inverse-list-combined message, with the same ordering and with corresponding method definitions, each component flavor receives the value which came from that flavor.

Here is a table of all the method types used in the standard system (a user can add more, by defining new forms of method-combination).

(no type) — If no type is given to defmethod, a primary method is created. This is the most common type of method.

:before
:after — These are used for the before-daemon and after-daemon methods used by :daemon method-combination.

:default — If there are no untyped methods among any of the flavors being combined, then the :default methods (if any) are treated as if they were untyped. If there are any untyped methods, the :default methods are ignored.

Typically a base-flavor (see page 23) will define some default methods for certain of the messages understood by its family. When using the default kind of method-combination these default methods will not be called if a flavor provides its own method. But with certain strange forms of method-combination (:or for example) the base-flavor uses a :default method to achieve its desired effect.

:wrapper — Used internally by defwrapper.

:combined — Used internally for automatically-generated *combined* methods.

The most common form of combination is :daemon. One thing may not be clear: when do you use a :before daemon and when do you use an :after daemon? In some cases the primary method performs a clearly-defined action and the choice is obvious: :before :launch-rocket puts in the fuel, and :after :launch-rocket turns on the radar tracking.

In other cases the choice can be less obvious. Consider the :init message, which is sent to a newly-created object. To decide what kind of daemon to use, we observe the order in which daemon methods are called. First the :before daemon of the highest level of abstraction is called, then :before daemons of successively lower levels of abstraction are called, and finally the :before

daemon (if any) of the base flavor is called. Then the primary method is called. After that, the :after daemon for the lowest level of abstraction is called, followed by the :after daemons at successively higher levels of abstraction.

Now, if there is no interaction among all these methods, if their actions are completely orthogonal, then it doesn't matter whether you use a :before daemon or an :after daemon. It makes a difference if there is some interaction. The interaction we are talking about is usually done through instance variables; in general, instance variables are how the methods of different component flavors communicate with each other. In the case of the :init message, the *init-plist* can be used as well. The important thing to remember is that no method knows beforehand which other flavors have been mixed in to form this flavor; a method cannot make any assumptions about how this flavor has been combined, and in what order the various components are mixed.

This means that when a :before daemon has run, it must assume that none of the methods for this message have run yet. But the :after daemon knows that the :before daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should "transmit" the information in a :before daemon, and the second one should "receive" it in an :after daemon. So while the :before daemons are run, information is "transmitted"; that is, instance variables get set up. Then, when the :after daemons are run, they can look at the instance variables and act on their values.

In the case of the :init method, the :before daemons typically set up instance variables of the object based on the init-plist, while the :after daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

Of course, since flavors are not hierarchically organized, the notion of levels of abstraction is not strictly applicable. However, it remains a useful way of thinking about systems.

1.13 Implementation of Flavors

An object which is an instance of a flavor is implemented using the data type **dtp-instance.** The representation is a structure whose first word, tagged with dtp-instance-header, points to a structure (known to the microcode as an "instance descriptor") containing the internal data for the flavor, and whose remaining words are value cells containing the values of the instance variables. The instance descriptor is a defstruct which appears on the si:flavor property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling messages, and information for accessing the instance variables.

defflavor creates such a data structure for each flavor, and links them together according to the dependency relationships between flavors.

A message is sent to an instance simply by calling it as a function, with the first argument being the message keyword. The microcode binds self to the object, binds the instance variables (as special closure variables) to the value cells in the instance, and calls a dtp-select-method associated with the flavor. This dtp-select-method associates the message keyword to the actual function to be called. If there is only one method, this is that method, otherwise it is an automatically-generated function which calls the appropriate methods in the right order. If there are wrappers, they are incorporated into this automatically-generated function.

The function-specifier syntax (:method *flavor-name optional-method-type message-name*) is understood by fdefine and related functions. It is preferable to refer to methods this way rather than by explicit use of the flavor-method-symbol (see page 15).

1.13.1 Order of Definition

There is a certain amount of freedom to the order in which you do defflavor's, defmethod's, and defwrapper's. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. It is considered important that not all the methods for a flavor need be defined in the same file. Thus the partitioning of a program into files can be along modular lines.

The rules for the order of definition are as follows.

Before a method can be defined (with defmethod or defwrapper) its flavor must have been defined (with defflavor). This makes sense because the system has to have a place to remember the method, and because it has to know the instance-variables of the flavor if the method is to be compiled.

When a flavor is defined (with defflavor) it is not necessary that all of its component flavors be defined already. This is to allow defflavor's to be spread between files according to the modularity of a program, and to provide for mutually-included flavors (see the :included-flavors defflavor option, page 21). Methods can be defined for a flavor some of whose component flavors are not yet defined, however in certain cases compiling those methods will produce a spurious warning that an instance variable was declared special (because the system did not realize it was an instance variable). In the current implementation these warnings may be ignored, although that may not always be true in the future.

The methods automatically generated by the :gettable-instance-variables and :settable-instance-variables defflavor options (see page 20) are generated at the time the defflavor is done.

The first time a flavor is instantiated, the system looks through all of the component flavors and gathers various information. At this point an error will be signalled if not all of the components have been defflavor'ed. This is also the time at which certain other errors are detected, for instance lack of a required instance-variable (see the :required-instance-variables defflavor option, page 21). The combined methods (see page 12) are generated at this time also, unless they already exist. They will already exist if compile-flavor-methods was used, but if those methods are obsolete because of changes made to component flavors since the compilation, new combined methods will be made.

After a flavor has been instantiated, it is possible to make changes to it. These changes will affect all existing instances if possible. This is described more fully immediately below.

1.13.2 Changing a Flavor

You can change anything about a flavor at any time. You can change the flavor's general attributes by doing another defflavor with the same name. You can add or modify methods by doing defmethod's. If you do a defmethod with the same flavor-name, message-name, and (optional) method-type as an existing method, that method is replaced with the new definition. Currently there is no good way to remove a method.

These changes will always propagate to all flavors that depend upon the changed flavor. Normally the system will propagate the changes to all existing instances of the changed flavor and all flavors that depend on it. However, this is not possible when the flavor has been changed so drastically that the old instances would not work properly with the new flavor. This happens if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance), or if you change the component flavors (which can change several subtle aspects of an instance). The system does not keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead it gives you a warning message, on the error-output stream, to the effect that the flavor was changed incompatibly and the old instances will not get the new version. The system leaves the old flavor data-structure intact (the old instances will continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data-structure, thus affecting the old instances that point at it.

One exception to this is that changes to defwrapper's are never automatically propagated. This is because doing so is expensive and the system cannot tell whether you really changed it or just redefined it to be the same as it was. (Note that the initial definition of a wrapper *is* propagated, but redefinitions of it are not.) See the documentation of defwrapper for more details.

1.13.3 Restrictions

There is presently an implementation restriction that when using daemons, the primary method may return at most three values if there are any :after daemons. This is because the combined method needs a place to remember the values while it calls the daemons. This will be fixed some day.

In this implementation, all message names must be in the keyword package, in order for the flavor-method-symbols (see page 15) to be unique, and for various tools in the editor to work correctly.

1.14 Entities

An *entity* is a Lisp object; the entity is one of the primitive datat/pes provided by the Lisp Machine system (the data-type function (see page 111 in the Lisp Machine Manual) will return dtp-entity if it is given an entity). Entities are just like closures: they have all the same attributes and functionality. The only difference between the two primitive types is their data type: entities are clearly distinguished from closures because they have a different data type. The reason there is an important difference between them is that various parts of the (not so primitive) Lisp system treat them differently.

A closure is simply a kind of function, but an entity is assumed to be a message-receiving object. Thus, when the Lisp printer (see sections 18.2 and 18.4 in the Lisp Machine Manual) is given a closure, it prints a simple textual representation, but when it is handed an entity, it sends the entity a :print-self message, which the entity is expected to handle. The describe function (see page 261 in the Lisp Machine Manual) also sends entities messages when it is handed them. So when you want to make a message-receiving object out of a closure, as described on page 7, you should use an entity instead.

Usually there is no point in using entities instead of flavors. Entities were introduced into Lisp Machine Lisp before flavors were, and perhaps they would not have been had flavors already existed. Flavors have had considerably more attention paid to efficiency and to good tools for using them.

[The rest of this section is not yet written. It would explain how to create entities, and how the defselect function is used to make a function that dispatches on its first argument at relatively high speed.]

1.15 Useful Editor Commands

Since we presently lack an editor manual, this section briefly documents some editor commands that are useful in conjunction with flavors.

meta-.

The meta-. (Edit Definition) command can find the definition of a flavor in the same way that it can find the definition of a function.

Edit Definition can find the definition of a method if you give

(:method *flavor type message*)

as the function name. The keyword :method may be omitted. Completion will occur on the flavor name and message name as usual with Edit Definition.

meta-X Describe Flavor

Asks for a flavor name in the mini-buffer and describes its characteristics. When typing the flavor name you have completion over the names of all defined flavors (thus this command can be used to aid in guessing the name of a flavor). The display produced is mouse sensitive where there are names of flavors and of methods; as usual the right-hand mouse button gives you a menu of operations and the left-hand mouse button does the most common operation, typically positioning the editor to the source code for the thing you are pointing at.

meta-X List Methods
meta-X Edit Methods

Asks you for a message in the mini-buffer and lists all the flavors which have a method for that message. You may type in the message name, point to it with the mouse, or let it default to the message which is being sent by the Lisp form the cursor is inside of. List Methods produces a mouse-sensitive display allowing you to edit selected methods or just see which flavors have methods, while Edit Methods skips the display and proceeds directly to editing the methods. As usual with this type of command, the editor command control-. is redefined to advance the editor cursor to the next method in the list, reading in its source file if necessary. Typing control-. while the display is on the screen edits the first method.

meta-X List Combined Methods
meta-X Edit Combined Methods

Asks you for a message and a flavor in two mini-buffers and lists all the methods which would be called if that message were sent to an instance of that flavor. You may point to the message and flavor with the mouse, and there is completion for the flavor name. As in List/Edit Methods, the display is mouse sensitive and the Edit version of the command skips the display and proceeds directly to the editing phase.

List Combined Methods can be very useful for telling what a flavor will do in response to a message. It shows you the primary method, the daemons, and the wrappers and lets you see the code for all of them; type control-. to get to successive ones.

Index